Flexible MIPS Soft Processor Architecture

by

Roberto Carli

B.S. Electrical Engineering and Computer Science, B.S. Management Sciences.
Massachusetts Institute of Technology, 2007

Submitted to the department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science
at the
Massachusetts Institute of Technology

May 2008

Signature of Author: _____

Department of Electrical Engineering and Computer Science
May 9, 2008

Certified by: _____

Cnristopher J Terman
urer
visor

Accepted by: _____

Arthur C. Smith
Professor of Electrical Engineering
Chairman, Department Committee on Graduate Theses

Flexible MIPS Soft Processor Architecture

by
Roberto Carli

Submitted to the department of Electrical Engineering and Computer Science

May 9, 2008

In partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

The flexible MIPS soft processor architecture borrows selected technologies from high-performance computing to deliver a modular, highly customizable CPU targeted towards FPGA implementations for embedded systems; the objective is to provide a more flexible architectural alternative to coprocessor-based solutions. The processor performs out-of-order execution on parallel functional units, it delivers in-order instruction commit and it is compatible with the MIPS-1 Instruction Set Architecture. Amongst many available options, the user can introduce custom instructions and matching functional units; modify existing units; change the pipelining depth within functional units to any fixed or variable value; customize instruction definitions in terms of operands, control signals and register file interaction; insert multiple redundant functional units for improved performance. The flexibility provided by the architecture allows the user to expand the processor functionality to implement instructions of coprocessor-level complexity through additional functional units. The processor design was implemented and simulated on two FPGA platforms, tested on multiple applications, and compared to three commercially available soft processor solutions in terms of features, area, clock frequency and benchmark performance.

CONTENTS

FIGURES

TABLES

1- INTRODUCTION

With the widespread success of smart phones and mp3 players, portable electronic devices are becoming increasingly complex, and currently represent the leading edge of digital hardware technology. To accommodate portability, research in hardware architecture is gradually shifting from very high performance general-purpose systems to dedicated low-power solutions, efficient and customized towards a specific task.

While dedicated hardware solutions deliver good results in terms of production cost, performance and power efficiency, developing such specialized systems makes engineering cost a principal factor in evaluating project feasibility. My research objective is to allow development of optimal dedicated hardware solutions without incurring in high case-by-case engineering costs.

My work contributes to the advancement of the REDSOCS project, currently under development at CSAIL. The aim of the project is to create interfaces for wireless, self-describing devices, which make their capabilities available to other machines over the network. These devices need to perform a wide variety of tasks, such as wireless communication and multimedia processing, locally and in hardware, by use of integrated system-on-a-chip (SOC) solutions. The integrated hardware solutions need to use a combination of input/output modules, high-speed and low-speed buses, and one or more central processing cores. The aim of such SOC solutions is to provide devices with the computational power required to perform the required tasks, while at the same time saving in production cost and power consumption by eliminating the flexibility and excess capabilities of a general-purpose solution. However, since different devices require different computational capabilities, a variety of specialized cores would have to be designed, thus making engineering cost a major limiting factor for development.

With the purpose of eliminating the contrast between hardware specialization and engineering costs, I designed and implemented a highly customizable out-of-order MIPS soft processor. While compatible with the standard MIPS ISA, the processor is based on a series of parallel, user-customizable functional units. By controlling these functional units through a set of parameters, a hardware designer (the user) can modify the hardware implementation of each instruction, including the internal degree of pipelining, as well as insert new functional units associated to custom instructions that are added to the processor ISA. Through customization, the user can generate a specific processor instance specially tuned for the needs of the particular system, without having to design the new core, and guaranteeing the required performance while minimizing area and power consumption. With several dimensions of customization, the hardware

designer is able to test, using an FPGA, which processor configuration works best for the required task in terms of performance/area/power tradeoffs, taking custom-design engineering costs out of the equation. Since the architecture complies with standard MIPS specifications, these core solutions may be used within a variety of implementations. Finally, the architecture may be used to develop a set of individually customized cores that share a common API: an interesting possibility for differentiated multi-core solutions.

2- Rationale and Feasibility Analysis

Giving higher flexibility to microprocessor solution is a problem that has been researched in the past. Dating back to the Intel 8087 for IBM PC (Palmer, 1980), the first mainstream solution to computational flexibility has been the insertion of a coprocessor unit. The specialized coprocessors are coupled with a specific Application Programming Interface (API) that specifies the interaction protocol between processor and coprocessor (Anderson, Svendsen, & Sohn, 1996), and allows users to write code which will take advantage of such extensions. The advent of reprogrammable logic and FPGAs introduced the notion of flexible coprocessors, whose internal functionality can be modified, while holding a fixed main processor and API (Hauser & Wawrzynek, 1997). With my thesis, I experiment a new approach to computational flexibility by proposing a new API, allowing the user to insert new functional units within the processor architecture.

In this section, I identify the main obstacles to a custom-made functional units architecture and study the feasibility of various methods to overcome them. In particular, I borrow and dissect various technologies from the high-performance computing domain ad examine their costs and benefit towards the realization of this new architecture. In evaluating technologies, I hold flexibility as my main architectural objective, while ensuring that an FPGA processor implementation keeps a good balance of performance, area and power consumption as typically required of integrated computing solutions.

The fundamental degree of freedom for this processor is the possibility to easily insert a variety of functional units that can be designed externally, either implementing a standard MIPS instruction or inserting a new, dedicated ISA instruction. However, if a custom functional unit (FU) is created to implement a complex, long-latency or multi-stage function, the effect on the final performance of the processor can be disastrous. This calls for a system that allows complex pipelined

functional units to coexist with regular single-cycle units without compromising performance. The easiest solution is a simple FU busy signal that can stall the processor pipeline until the FU has completed its job. However, this system does not allow for any parallelism in the operation of the various FU.

The solution I use is a modified implementation of Tomasulo's algorithm (Tomasulo, 1967), which allows functional unit parallelism and Out-of-Order execution while retaining in-order issue and commit of the instructions. This system also allows continued operation for Write-After-Write (WAW) and Write-After-Read (WAR) data hazards. The algorithm is modified for improved handling of memory instruction and for retaining in-order issuing and committing while executing out-of-order.

The basic principle behind this solution is register renaming: the source and destination registers become pointers to either the register value, if available, or to the functional unit that is going to produce it. This added degree of register flexibility requires a space to hold the extra information, which is provided by the modules called reservation stations, each associated with a different functional unit, and by a register status table that keeps track of available and pending data.

Register Status

The register status table, effectively a bookkeeping table for register renaming, holds information about all the architectural registers. For each register, it holds either a zero (when the value is available in the register), or it points to the reservation station which is assigned going to produce the value.

Reservation Stations

Each functional unit needs to be associated to one or more reservation stations, holding information about its current operation. These stations represent a decentralized way of having each FU hold information about its operation, so that instruction issue can happen sequentially, and each FU can independently handle data dependency issues. From an algorithmic perspective, a pipelined FU that can concurrently process multiple instructions at different stages is equivalent to multiple single-instruction FUs, each of which needs its own reservation station. Thus the registers and instructions need to always refer to reservation stations as opposed to FUs.

A reservation station is a data structure that holds relevant information about the operation that the associated functional unit has to complete (Hennessy & Patterson, 2006). These are the fields that characterize the state of a reservation station:

- **Busy:** A signal indicating the status of the FU. When busy, the functional unit cannot receive a new instruction.
- **Op**: When a single FU can perform more than one operation (e.g. ADD/SUB), Op holds information about which operation needs to be performed.
- **Qj, Qk:** Fields used for pointing to unavailable operands. They point to the reservation station which is assigned to produce the operand value. If instead the operand is available, they hold a value of zero.
- **Vj, Vk:** Fields holding the value of the operands. They can only be read if the corresponding Qj, Qk are zero, indicating that the operands are available. When both Vj and Vk are present the reservation station issues an operation to the FU.
- **A:** Immediate value. Never points to a reservation station since immediate values are provided directly by the instruction. Sixteen-bit immediate values are used for address computation in branching and memory instructions, and as constant arithmetic operands.

BASIC OPERATION ALGORITHM

With reservation stations holding the renaming information, instruction issuing becomes a matter of searching for and loading available reservation stations that can execute the instruction, as well as interacting with the instruction memory and managing the pc logic. Figure 1 outlines the action and bookkeeping steps of the algorithm. Figure 2 illustrates a possible processor state under this method.

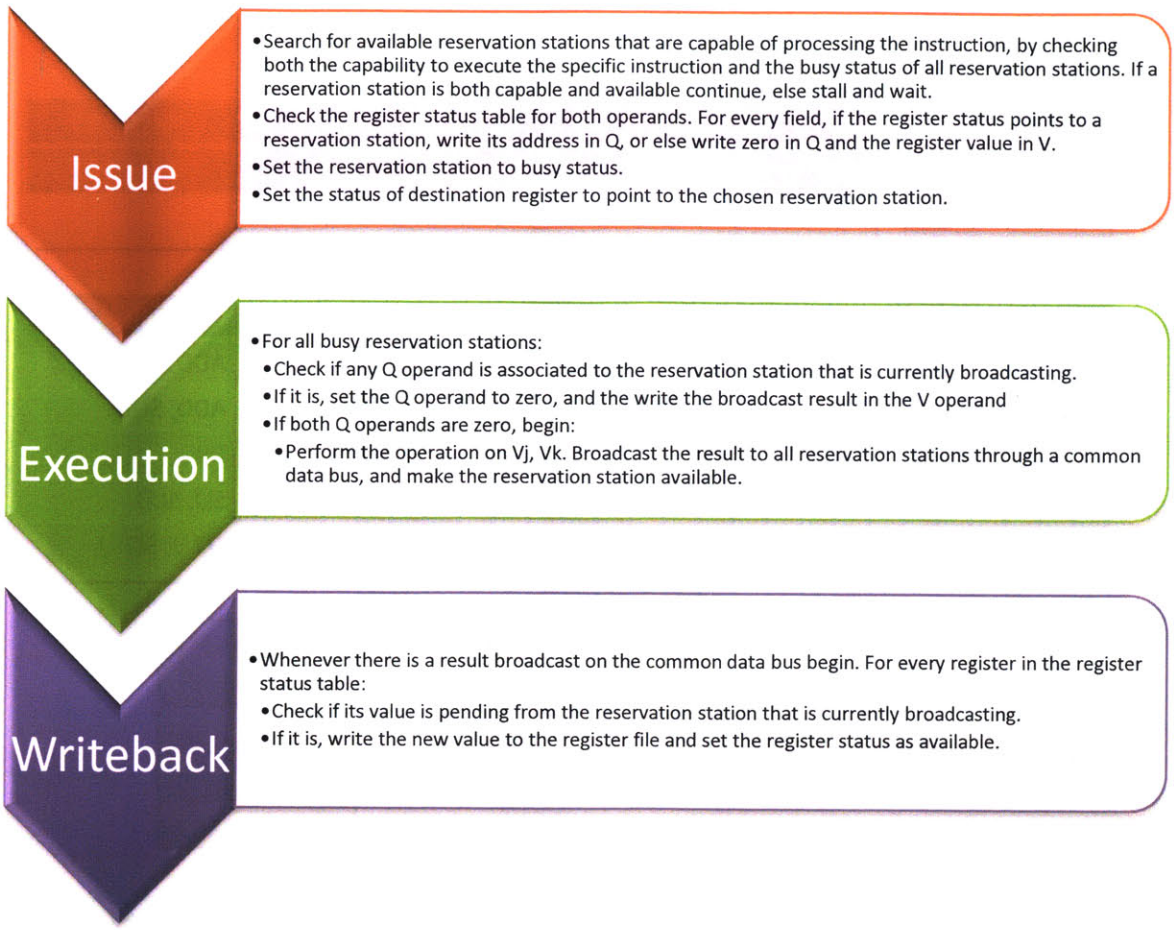

Figure 1: Overview of Tomasulo's algorithm for out-of-order execution. The description does not include the arbiter mechanisms described further down, designed to resolve Common Data Bus conflicts and ensure in-order instruction commit.

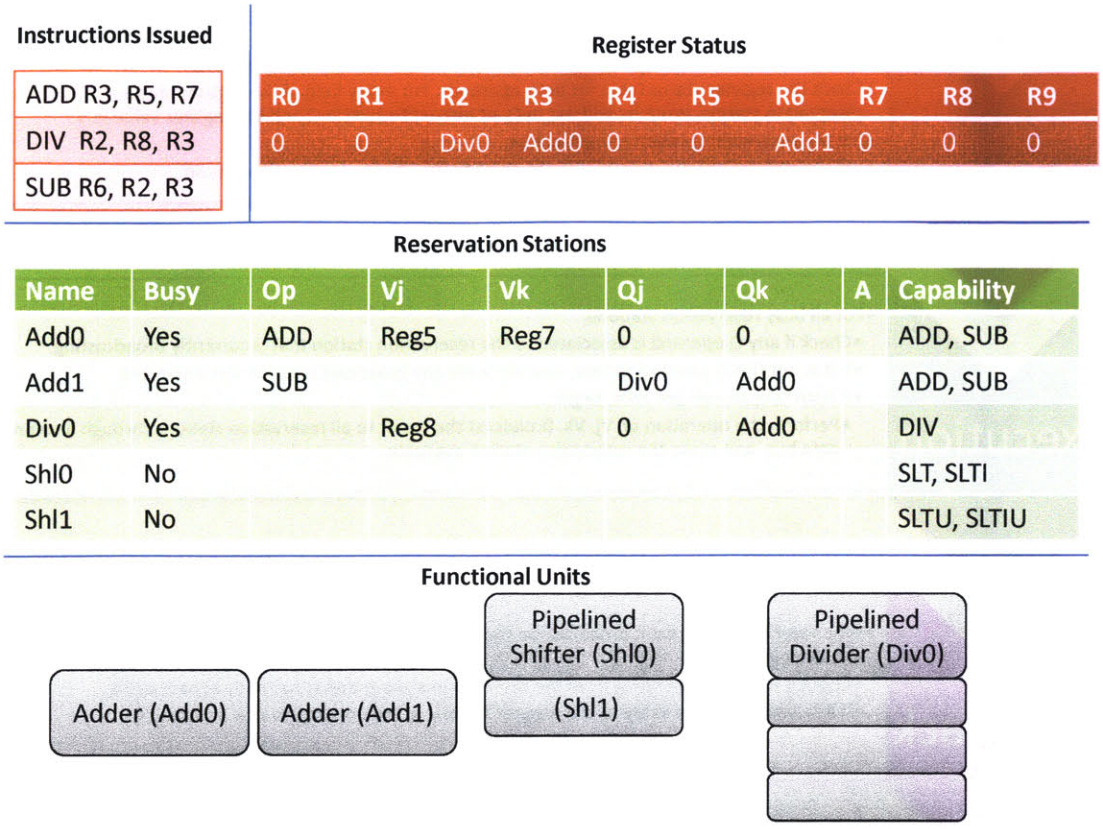

Figure 2: Snapshot of a possible processor state, illustrating the handling of RAW data hazards. The processor has two adders (Add0, Add1), a pipelined two-stage shifter that can operate on two instructions at a time (Shl0, Shl1), a circularly pipelined divider that can operate on one instruction at a time (Div0). In this case, while the ADD instruction (Add0) is completing, the DIV instruction has been issued (Div0) and is waiting for Add0. The SUB instruction (Add1) has also been issued, and is waiting for both Add0 and Div0. At this point, if the next instruction finds no data dependencies and a free functional unit (i.e. SHL R4, R5, R8), it can be issued and executed immediately.

BRANCHING

Out-of-order execution implies the issuing of new instructions when the result of previous ones is still unknown. In the case of branching, the result determines which instruction needs to be issued next. For out-of-order processors, this situation implies a fundamental choice on whether to continue speculative execution after a branch is issued and before its result is known.

The general approach to supporting speculative execution is the introduction of a "commit" stage, where the instruction result is written into registers only if the result of the branch preceding the

instruction has been computed. Supporting instruction commit implies the introduction of additional bookkeeping registers, usually implemented with a Re-Order Buffer (ROB) algorithm.

A more straightforward solution to branching in out-of-order processor consists of simply stalling instruction issue at a branch until the branching instruction is resolved, which can be associated to the *busy* bit of the branch FU reservation station.

Feasibility Analysis for Speculative Execution and Branch Penalty

Support for speculative execution has become the standard practice for modern high-performance processors. However, while a ROB implementation does not increase processor latency, it implies a very high cost in new hardware. In particular, the introduction of a whole new set of registers is very expensive on FPGA implementations, not to mention that speculative execution requires the introduction of branch prediction hardware, further complicating the overall architecture.

The cost of implementing Tomasulo's algorithm is justified by the possibility of inserting custom pipelined functional units for high architectural flexibility and improved performance. On the other hand, for the overall performance of the processor, the benefits of speculative execution depend on average branch penalty and branch frequency.

A measurement of branch penalty is given by the average number of cycles "lost" in a piece of code with a branch instruction as opposed to the same code without any branching. In this processor, the penalty depends on several factors. First, the branch penalty increases with a higher degree of parallelism, thus it would increase in a processor with a very large number of custom FUs. However, the introduction of custom FUs is aimed towards flexibility rather than high performance through parallelism, so it is reasonable to assume that the typical user-generated processor would not feature a very high number of custom FUs. Second, branch penalty increases with superscalar issuing, thus reducing the marginal benefit of a superscalar implementation. Third, when dealing with out-of-order execution, it is fundamental that the branch instruction is not waiting on pending data, so that it can be resolved quickly. In typical programs, most branches are the result of loop unrolling (i.e. *for* loops), most likely depending on a simple counter value rather than on the result of more complex operations. In this case, most branch instructions would find their operand quickly available. This general speculation, however, depends on the typical benchmark code that the processor will have to execute.

Another factor determining the feasibility of speculative execution is branch frequency, which is highly variable depending on the type of code executed, yet typically ranging between 15-30% of total instructions (McFarling & Hennessey, 1986). Branch frequency can be a major factor in

reducing instruction-level parallelism (ILP) in code execution, and benchmarking code can give a good indication of such penalties.

Summarizing, branch prediction is a common and useful technique, yet a mechanism for misprediction recovery would add an excessive complexity penalty to the architecture. A high branch frequency can severely limit the maximum ILP obtainable by an out-of-order architecture, yet this cost is bearable since the architecture aims at maximizing flexibility over performance.

Feasibility of Superscalar Issuing

With data dependencies handled by reservation stations, and instruction issuing independent of WAW and WAR hazards, it seems natural to try issuing multiple instruction per cycle. However, while logically easy, superscalar issuing implies many hardware complications.

The processor writeback implementation is based on a Common Data Bus, feeding results to all reservation stations, so that newly obtained register values are broadcasted for immediate update, thus one element to be considered is the impact of a larger, more complex Common Data Bus. Superscalar issuing implies the use of a larger CDB for two reasons. First, for a superscalar approach to become advantageous the processor must have a high number of FUs to increase the chance of being able to perform multiple issuing. The introduction of more FUs will increase the complexity of the CDB, since every FU can both write to and read from it. Second, by simple queuing reasoning, a processor cannot issue an average of more than one instruction per cycle unless it is also able to commit multiple instruction per cycle, or else the number of in-flight instructions would grow indefinitely. This capability would require the instantiation of multiple CDBs, again increasing hardware complexity. Finally, FPGA interconnections are generally efficient for high-throughput networks, yet much weaker for high-interconnectivity setups (Kapre, et al., 2006). A more complex CDB interconnecting all functional units and reservation stations would therefore pose an even higher hardware cost when part of an FPGA implementation

Additionally, when operating without speculative execution, superscalar issue is stalled by the presence of branching instructions. Therefore, frequent branching limits the usefulness of superscalar issuing.

Exception Handling

While the issuing of instructions is sequential, out-of-order execution does not guarantee sequential completion of instructions. Therefore, if an interrupt happens at a random point in the code, finding an appropriate state in which to stop execution is a nontrivial task. The typical solution in modern architectures consists of using the ROB to keep track of committed

instructions, and rolling back to the latest commit point in case of an interrupt, just like would happen for a mispredicted branch. However, only speculative processors feature a ROB that can be used for the purpose.

For non-speculative out-of-order processors, there are two ways of handling interrupts. The first solution implies the implementation of a book-keeping register set holding and forwarding instruction results, while committing them in-order to the register file. At an interrupt, the book-keeping table can be cleaned and execution blocked at the latest committed instruction. This approach is logically slightly easier than the implementation of a full ROB, however it still implies the expensive introduction of a new register set. To achieve full parallelism, the register set needs to hold one entry per reservation station, nearly doubling the hardware cost of out-of-order execution when implemented on FPGAs. Practically, this solution is unfeasible unless coupled with a full ROB and speculative execution.

The second solution is a more straightforward approach. Similarly to the branching solution, the issuing can stall and the interrupt can wait until all reservation stations are empty. The feasibility of this solution increases when there are fewer multi-stage functional units (differently from branching, the number of FUs is irrelevant, the limiting factor is the slowest FU), and when interrupts are relatively infrequent. Again, benchmark applications can provide estimates of interrupt frequency.

ARBITER MODULE

The adopted solution is a memory-based module functioning as a writeback arbiter. This module is designed to address three problems.

1. FU Errors and in-order commit

Processor interrupts are not only generated externally, but can also come as a result of an error signal given by a functional units, a divide-by-zero error being the typical example. In this case, the instruction execution has to interrupt precisely at the error-generating instruction, even though subsequent instructions may already be in-process within their functional units. As discussed earlier, however, in a system without a reorder buffer rollbacks are not possible. In this case the only solution is to sacrifice some performance to implement in-order instruction commits. There must be a system that controls CDB writeback so that FUs complete in the original instruction order.

2. Destination register, avoiding content-addressable memory

The original definition of Tomasulo's algorithm describes the Common Data Bus as carrying the

result value, together with a tag identifying the FU that produced it. This tag is matched against all outstanding *"Q"* fields in the reservation stations and in the register status table, so that every field waiting for the result of that FU can be updated.

From a hardware prospective, this design implies that the register status table be implemented as a content-addressable memory, where the register that needs to be written has to be found by matching the FU tag with each entry value. This functionality is incredibly expensive and should be avoided unless strictly necessary. Another reason to avoid this solution is that the inverse lookup latency is added to the writeback critical path, as a destination register is needed before the register file can be updated.

The way to avoid content-addressable memory is to memorize the destination register of every functional unit currently operating, so that the register value needs not be retrieved from the register status table

3. CDB management

Finally, while in-order issue guarantees that at most one instruction per cycle is issued, the variable latency of FUs implies that more than one result may be ready for writeback at every clock cycle. Since there is only one CDB available, the arbiter manages and prioritizes writeback order from all FUs.

Arbiter specifications

The introduction of an arbiter module solves the three issues described by implementing the following algorithm. Also, it provides functional units with a free-to-write authorization signal to use the CDB.

Issue

If a new instruction is issued, store the instruction's destination register and reservation station tag as a new FIFO entry

Writeback

Every clock cycle, the arbiter reads the first FIFO output (earliest in-flight instruction).

If the reservation station described by the tag is ready, give it a free-to-write signal, and use the destination register as a write address for the register status table. Delete this entry (FIFO read enable).

If the chosen reservation station is not ready, wait and do not let any FUs use the CDB.

With this functionality, it is possible to see how in-order commits are ensured, at most one FU per cycle can use the CDB, and the destination register is provided at writeback, making a content-addressable register status table unnecessary.

Reservation stations are still updated by inverse lookup. However, this does not imply additional hardware: all reservation stations fields need to be fully updated in a single clock cycle at instruction issue, thus cannot be implemented as single or dual-port memories, but rather as collections of independent registers. In this case, adding two comparators for the *Q* fields listening to CDB values does not add significant hardware cost.

SUMMARY

From the analysis of superscalar issuing, exception handling and speculative execution, it is evident that the methods to improve performance of an out-of-order processor tend to be effective only when used together. Summarizing the feasibility analysis, it can be seen that superscalar issuing implies higher branch penalty, which needs to be fixed by speculative execution, which in turn requires branch prediction hardware. On the other hand, there is the possibility of using simple solutions for branch and exception handling, which slightly decrease the effectiveness of out-of-order execution but come very cheap in terms of hardware resources.

The introduction of additional hardware for speculative execution does not fit the scope of this project. While expensive, out-of-order execution dramatically increases flexibility by allowing simple operations to coexist with complex custom functional units. On the other hand, book-keeping techniques for precise exception handling and speculative execution only influence performance under frequent branching/exception conditions, while nearly doubling the additional hardware cost on FPGA implementations.

3- FUNCTIONAL SPECIFICATIONS

In order to increase customization freedom, each functional unit and reservation station is implemented as a separate module, rather than being organized in table form. Another priority was to simplify the interface requirements for the functional units, as these are the components that can be modified externally, without requiring knowledge of the processor architecture.

Register Status Table

The register status table is implemented as a 4-port memory of size [number of registers x log2(number of reservation stations)]. Every cycle it can perform 2 reads and 2 writes:

2 reads from the Issuer, to read the status of the rs and rt fields of the instruction

1 write from the Issuer to specify that the value of the target register is no longer known, as it will be produced by the chosen reservation station (Register Status [rd_issuer] <- reservation station tag)

1 write from the common data bus to communicate that the register value is now known (Register Status [rd_arbiter] <- 0)

The write from the common data bus writeback can be performed only under two conditions:

It does not conflict with current write from the Issuer. If it conflicts (same target register), discard the write.

Before writing, check that the Register Status entry is still looking for the reservation station trying to write (current status of register = common data bus tag). If not, it means that another instruction with the same target register has been issued while the reservation station was at work, thus the register value is still pending, thus the write should be discarded.

Register File and Memory

The register file can be implemented regularly, with 2 read ports and 1 write port. The memory is implemented as a single module with 2 read-ports and 1 write-port, incorporating both instruction and data memory.

Issuer

The Issuer is be implemented as an separate module. The hardware requirements are:

Read the instruction from instruction memory

Produce a stall signal to stop pc from incrementing if (a) there is no available reservation station for processing the instruction or (b) the instruction is a branch or (c) the instruction is an interrupt. To preserve consistency, in case of interrupts and branches, the processor stalls until all previous instructions have been committed.

Read the register file and the register status table in parallel for rs and rt. If the register status

produced a value, use it, or else use the register file value.

Broadcast values for all fields (busy,op,Vj etc.) to all reservation stations, but only assert a write enable for the appropriate station.

Feed the tag of the chosen reservation station and the destination register to the arbiter.3 (*to_arbiter_dest, to_arbiter_tag*).

COMMON DATA BUS

The CDB is composed of registers holding writeback information. The registers act as a writeback pipeline, and their content is broadcasted to the register file and register status tables, for register update, as well as to all the reservation stations, that monitor the CDB data for results coming from their Qj, Qk fields.

The CDB delays and broadcasts the following fields:

Destination (*from_cdb_dest*): The target register for the writeback result. Originally produced by the arbiter (*from_arbiter_dest*).

Tag (*from_cdb_tag*): Identifier for the reservation station that produced the result. Originally produced by the arbiter (*from_arbiter_tag*).

Data (*from_cdb_data*) : The new register value. Originally produced by one of the FUs. The appropriate FU output (*fu_chosen_output*) is selected in the previous stage by the arbiter tag, indicating which reservation station had to commit.

Write-enable (*from_cdb_we*) : It indicates whether the CDB data is valid. While tha data is always taken from the FU whose turn it is to commit, the write enable must be high only when the FU has completed operation. This signal originally comes from the chosen reservation station, which asserts the write enable when the FU has produced the correct result.

RESERVATION STATIONS

Except for the special cases of memory and branch instructions, all reservation stations are equal, and operate in the same way regardless of the functional specifications of their attached FUs. In particular, the RS can be in one of three states:

IDLE/WAITING FOR INSTRUCTION

When the issuer loads an instruction (write enable is high), look at all the operands:

If both operands are ready (Qj, Qk = 0), load all fields from the issuer and begin operation by starting the FU (raise *fu_start*)

If one or more operands are missing, check the CDB broadcast before loading the fields. If one of the missing fields (tag = Qj or Qk) is being broadcasted (cdb_we = 1), update the broadcast instead of the issuer value. If, thanks to the CDB, both fields are ready, begin operation.

Assert the busy signal.

WAITING FOR OPERANDS

Check the CDB broadcast. If one or both the missing fields (tag = Qj or Qk) are being broadcasted (cdb_we = 1), update the value.

If, with the current CDB broadcast, both operands are now ready, begin operation immediately.

WAITING FOR WRITEBACK

If the result is ready (*fu_ready)*, monitor the arbiter authorization (*arbiter_free_to_write)*. This signal will be asserted by the arbiter when the RS is the next due to commit (*from_arbiter_tag* = RS number). If the FU is ready and the authorization has been received:

Write the Fu result to the cdb (assert *to_cdb_we*)

Set the busy signal to zero.

FUNCTIONAL UNITS

Functional Units are the key components of the processor, since it is through the insertion and customization of functional units that the processor achieves its feature characteristic of flexibility. Ideally, functional units need to be:

Standardized in I/O: so that reservation stations can obey a single communication protocol with all FUs

Internally Customizable: each FU should be able to operate freely, without any constraint on internal components, timing requirements, or limitations on pipeline stages.

Architecture-Independent: of all the processor's components, the custom FUs are to be coded by the user/designer, who shall not be forced to learn about the FU requirements within the processor architecture. Ideally, the user should only specify the operational functioning, together with some characteristics of the new FU (such as the degree of internal pipelining). A FU

template should take care of analyzing these characteristics and generate the appropriate internal communication logic.

Fully Utilized: if a FU has many pipeline stages, yet can intake a new operation at each cycle (i.e. it is not "circularly pipelined"), it should have the possibility of processing, if needed, several in-flight instructions at one point in time

The architecture satisfies these requirements through the following mechanisms:

Standardized I/O: each FU is limited to a 3-input (*op1,op2,a*) 1-output (*result*) operation, whose details are covered in a later section (Figure 5, Figure 6). If capable of performing multiple operations (i.e. ADD/SUB) the Functional Unit can choose the appropriate one through a 4-bit op-code signal.

Internally Customizable: The FU template has a space where the user/designer can instantiate any internal logic, combinational or registered, as long as it wired to the 3-input 1-output framework of the FU.

Architecture-Independent: To interface with other components, the FU uses two communication signals (*fu_start, fu_ready*). Such signals, however, are managed by a standardized internal logic, which only uses as an input parameter the number of internal pipeline stages, and ensures correct communication. This way, the user/designer is not concerned with the architectural and timing requirements for the functional unit.

Full Utilization of Functional Units

If a FU internal logic can process several in-flight instructions, a special FU shell with attached controller is instantiated. This FU connects to as many reservation stations as the number of possible in-flight instructions, thus being seen by the processor as a group of independent one-operation-at-a-time FUs. With this method, it is possible to have a single FU shared by as many reservation stations as the maximum number of in-flight operations the FU can process. To accomplish this, the controller has to deal with the following tasks:

Operation intake: accept the *fu_start* signals from all the RS connected to the FU. When a RS issues a new *fu_start*, save the operands, the opcode, and an internal tag pointing to the specific RS. Solve conflicts with a priority table, and delay the *fu_start* of the RS which does not have precedence.

Pipeline bookkeeping: At every cycle, while the new operands go down the FU pipeline, keep a separate delay-pipeline for op-codes to ensure that, if different operations are performed at different stages, each stage reads the correct op-code. Also, delay the issuing RS tag by as many

delay slots

ARBITER

Based on a simplified FIFO structure, the arbiter keeps track of the instruction issuing order, and ensures in-order commit of instructions. Whenever the issuer processes a new instruction, it communicates to the arbiter the reservation station in charge and the destination register, which may be unspecified for instructions not writing to the register file. The destination register field avoids the necessity of implementing content-addressable memory: in absence of this field, on instruction commit the register status table would have to check which register is waiting for the given reservation station before writing the new value. The issuer ensures correct signaling of new instructions to the arbiter even in branching situations, when the processor is stalling.

On the output side, the arbiter broadcasts information on the next reservation station that has to commit, and the destination register. It also gives an authorization signal to the correct reservation station which, if the functional unit is ready with the result, will trigger an instruction commit and a write to the CDB.

LOAD/STORE

Load and Store instructions are processed by a special reservation station and connected functional unit. Since store instructions do not write to the register file, the load/store reservation station allows store instructions to commit to the arbiter without having to broadcast a result over the CDB.

The load/store functional unit features additional I/Os for interacting with the data memory. For algorithmic consistency, all address computations happen within the FU, as opposed to instruction issue, and the functional unit only receives register and immediate values. This happens since, given the architecture, the target address may depend on registers whose value is not available at issue time. When the start signal is asserted, the FU performs address computation and issues a read or write to a synchronous data memory.

BRANCH/JUMP

Another special reservation station and functional unit pair is used for all branching and jump instructions. As for the Load/Store, the reservation station differs by separating instruction commit from register file write back, since most branching instructions (except for JAL/JALR) do not affect the register file.

As for the Load/Store case, and for the same reasons, both target and branch condition

computations are performed within the FU as opposed to issue stage. To perform target computation, the FU is provided with the PC value as an additional input. The issuer ensures that the FU is not provided with the current pc, which might have changed, but rather with the correct pc value associated with the instruction.

In addition to writing to the CDB (for JAL/JALR instructions), the branching FU provides its result to the issuer. The branch result takes the form of a target address for the new pc, a *branch_taken* value providing the result of the branch condition, and a *branch_committed* that allows the issuer to recover from stalling by moving to either the new pc or pc+4.

Delay Slot Implementation

For increased performance, some MIPS architectures require that the instruction after a branch, called the delay slow, be executed before the branch is taken. While the delay-slot method is outdated, I decided to implement it as an optional, parameter-regulated feature, to ensure higher compatibility with different MIPS implementations. To implement delay slots, the issuer delays the stall after a branch, so that the next instruction is issued before stalling and waiting for the branch result.

When delay slots are enabled the issuer also ensures a defined behavior in the complex case of adjacent branches. While most MIPS architecture have undefined behaviors for adjacent branches, the issuer ignores the result of the first branch while prioritizing the branch positioned in the delay slot.

HIGH-LEVEL STRUCTURE

Figure 3 illustrates a high-level perspective of the processor implementation.

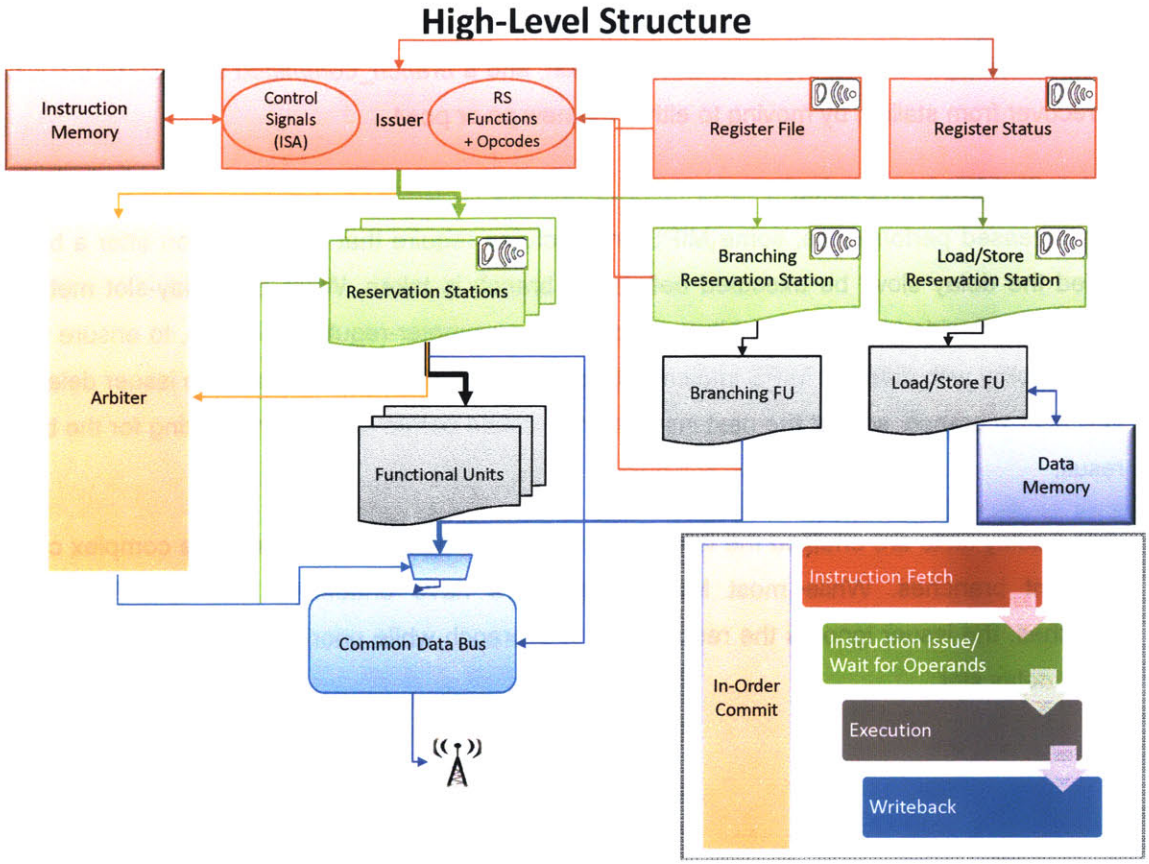

Figure 3: High-Level view of the processor implementation, representing all modules and the main interactions. Conveying the idea of flexibility, standard reservation stations and functional units are represented as a list, extensible and customizable by the user. Also, the user can modify or extend the set of control signals representing the ISA, thereby defining a new instruction type. As explained in the bottom-right portion, the color-coding pictures the pipelining structure, which consists of a baseline 4-stage operation with a flexible execution stage, where different functional units can be defined with different degrees of pipelining. Transcending from the pipelining structure, the arbiter regulates Common Data Bus access and ensures in-order instruction commits. Finally, the Common Data Bus broadcasts instruction results, which are monitored by the register file, register status table and all reservation stations.

4- CUSTOMIZATION

This section illustrates the details of the processor's user interface, and describes the user-exposed parameters that regulate customization.

CUSTOMIZABLE COMPONENTS

The flexibility of the processor architecture resides in the possibility to extend and modify the instruction set as well as the functional units. Figure 4 is an overview of the internal parameters that allow a user to define new functional units as well as new instructions.

For each new functional unit, the user needs to specify a set of parameters that the architecture will use. First, the instruction capability is a list of instructions that the functional unit is capable of executing. For each of these instructions, the user must specify an opcode that will be fed to the functional unit to recognize which instruction is being requested. For correct pipelining configuration, the user must specify the number of clock cycles that the functional unit will take to execute the instruction, as well as the Verilog code describing the execution logic.

When inserting a new instruction, the user must specify three parameters. A unique instruction name, which will be used by the architecture, a 32-bit instruction signature, which will allow the issuer to recognize the instruction, and a set of control signals to specify the correct operands and destination for the functional unit.

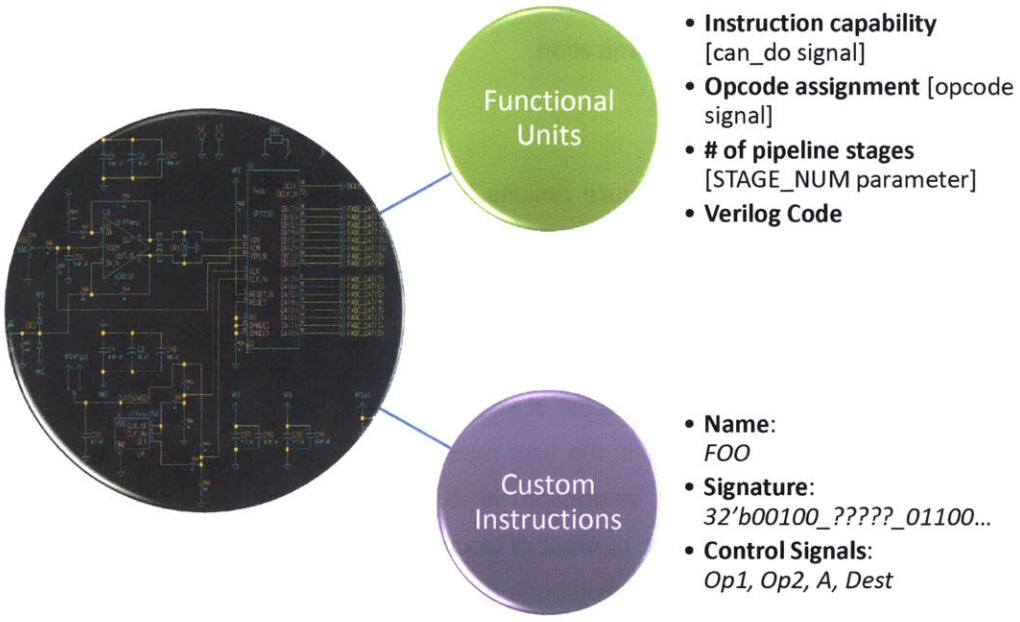

Figure 4: High-level overview of processor customization. The user describes new functional units and custom instructions, which are automatically implemented in the architecture of the generated core. In particular, the user must specify a set of architectural parameters for the additional components, as well as Verilog code describing the execution logic of new Functional Units.

CUSTOM FUNCTIONAL UNIT

Figure 5 offers a detailed example of the description required to specify a new, user-generated functional unit. In particular, the user needs to specify the following parameters

Instruction Capabilities: A list of all instructions the FU is capable of executing. Instructions should be referred by standard names in the MIPS ISA, or by the new names of custom instructions added to the ISA.

Opcodes: For all instructions in the capability list, the user must specify a 4-bit opcode that will be given to the Functional Unit for recognizing the required instruction.

# of Pipeline Stages: Each FU must declare its internal degree of pipelining. Purely combinational FUs must use zero for this parameters. While there is no limit on the number of pipeline stages allowed, such number of stages must hold for all instructions the FU is capable of processing. For a number of pipeline stages N, the architecture requires that the Functional Unit produce a valid result N cycles after valid operands have been provided. The operands will remain constant for the entire duration of the execution and, after N cycles, the result must stay valid as long as the operands remain valid. The user need not have knowledge of when the operands become valid, as these details are handled by the internal architecture.

Verilog Code [with constraints]: The user must specify Verilog code describing the execution logic, while observing a few constraints. The operands will come as signals *op1* [32-bit], *op2* [32-bit] and *A* [16-bit], as well as the *opcode* [4-bit] describing the kind of instruction requested. According to the relevant instruction, the FU might not utilize all input signals. The result must be produced over the Verilog wire *result* [32-bit] and assigned through combinational logic (*assign* statement). The keywords *clock, reset, start, ready* are reserved and may not be declared, yet *clock* and *reset* may be monitored as inputs. The signals *start* and *ready* should not be observed, since their timing specifications vary with the architectural role of the functional unit, thus referencing them may cause incorrect behavior at times. Constraints aside, the user is allowed to declare any signals or registers required for specifying internal logic.

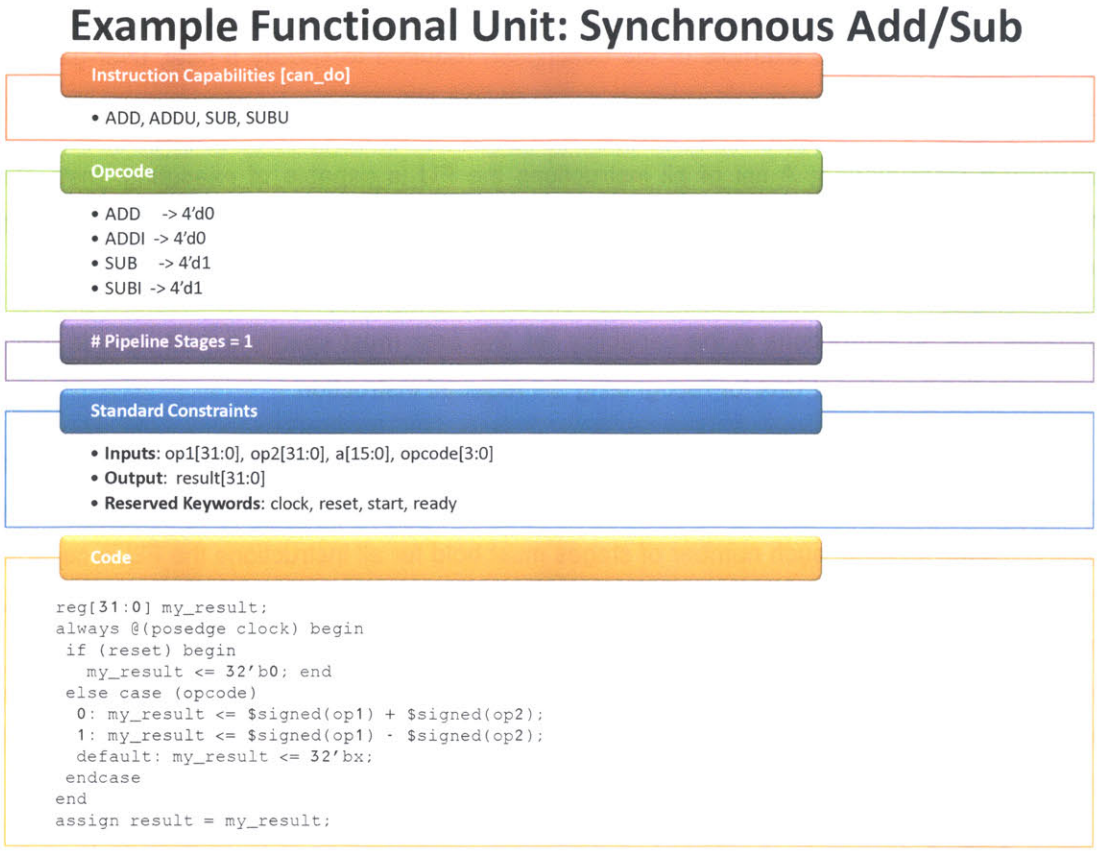

Figure 5: Description of a single-pipeline, synchronous functional unit, capable of performing signed Add and Subtract. First of all, the FU specifies a capability list of the instructions it can execute. Second, it specifies an opcode for each instruction.

In this case, notice that ADD/ADDI and SUB/SUBI have the same opcode. This is due to the processor's control signals choices, and will be further explained in the next figure. Within the ISA, the sign extension of the immediate can be performed before issuing. In this case, it can be deduced that the FU is always expecting the appropriate second operand in *op2*, whether it be Reg[Rt] (for ADD/SUB) or a 32-bit, sign-extended *immediate* (for ADDI/SUBI). Alternatively, the architecture allows the user to change the control signals for ADDI/SUBI in order to receive a 16-bit *immediate* in the *a* field, and performing the sign-extension within the functional unit.

Next, the user specifies the number of pipeline stages required by the functional unit, with zero corresponding to combinational execution. In this case, synchronous execution implies a pipeline value of one, as the result is ready one clock cycle after the operands become valid.

The next field simply reminds the user of the constraints on the Verilog code specifications. In particular, the input/output signals and the reserved keywords.

Finally, the user provides a Verilog description of the execution logic. In this example, the execution is synchronous, yet the result must be assigned to a combinational *result* wire, therefore the user declares an internal *my_result* register which will hold the synchronous result. The user-supplied code reads the *clock* and *reset* signals to setup the synchronous logic, and references to *opcode* within a case statement in order to perform the required operation on *op1* and *op2*. In this case, the input *a* is never needed. Finally, the *result* wire is assigned to the synchronous result.

From an overview of the Verilog code, it can be seen that (1) A correct result is produced 1 cycle after receiving valid inputs; (2) The result stays valid as long as the inputs stay valid; (3) No reserved keywords/signals are redeclared; (4) The user does not reference to the *start* or *ready* signals. The code therefore complies with the architectural specifications.

CUSTOM INSTRUCTION SPECIFICATION

Figure 6 is an example of the specifications for inserting a custom instruction in the processor's ISA. The following parameters are required for insertion:

Name: The new instruction needs a unique name, by which it will be identified within the architecture. While the user has freedom of choice, a two to four-letter capitalized name would fit organically with the MIPS architecture.

Instruction Signature: The user must specify a way for the architecture to uniquely recognize the instruction. The user specifies the signature by a 32-bit sequence of values one, zero or ? (to signify don't-care). This description must not overlap with any other instruction in the ISA, and must be compatible with the instruction format used. For example, for an I-type instruction using the immediate, the signature should have the value "?" for bits 16:0, as it should be recognized for all values of the immediate (bits 16:0 of the instruction).

Control Signals: Finally, the new instruction should specify its control signals for assigning the correct operands and destination values. It is important to remember that, while the format is not explicitly specified, the choice of control signals must comply with one instruction format type. For example, an instruction should not use both Rd (R-Type format) and Immediate (I-Type format), since these two values are not independent, they overlap on bits 16:12 of the instruction. There are four values to assign:

Op1: The first 32-bit operand. Choices between zero, Reg[Rs], a zero-extended shift amount value [bits 10:6 of the instruction], or x (unspecified / don't-care).

Op2: The second 32-bit operand. Choices between Reg[Rt], zero-extended immediate, sign-

extended immediate, zero-extended target [bits 25:0 of the instruction], or x (unspecified / don't-care).

A: 16-bit operand. Choices between zero, immediate or x (unspecified / don't-care).

Dest: 5-bit address specifying the destination register. Choices between R0, Rd, Rt, or R31.

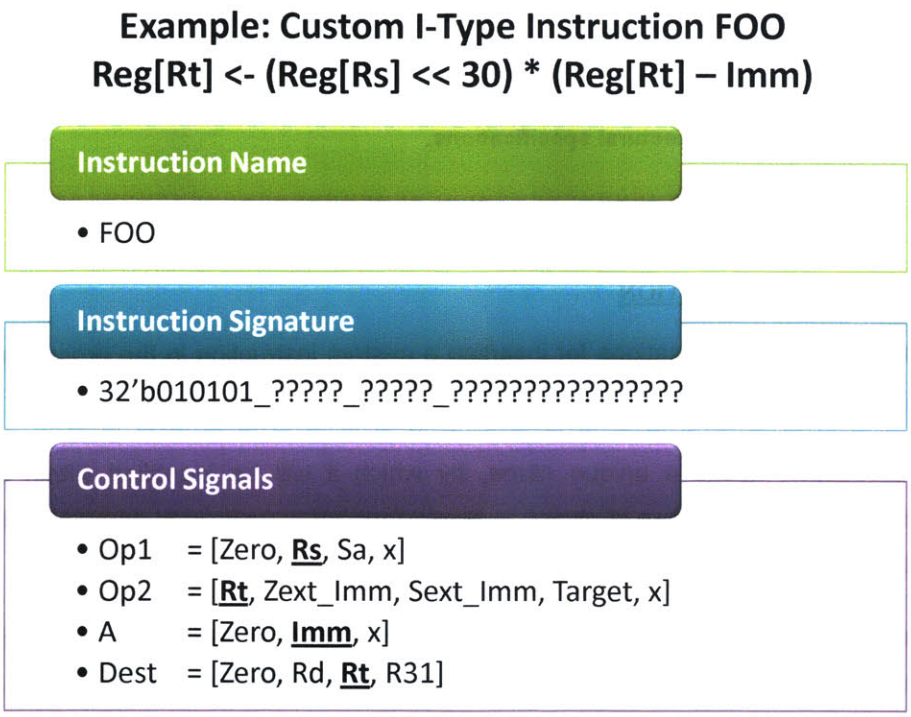

Figure 6: Specification of a custom instruction[1]. First, the unique name FOO is specified for the instruction. Second, a signature is given. This signature must not overlap with other instruction (in this case, it should have a unique *op* field), and must not specify constraints on operands (in this case, Rs, Rt, and immediate are all left as don't-care question marks). Finally, control signals are chosen from a list of possibilities. In this case, the functional unit will receive the required operands Reg[Rs], Reg[Rt] and Imm respectively through Op1, Op2 and A. Rt is set as the destination register. Finally, it is important to notice that *Rd, sh_amt* or *target* are not used, as they would overlap in the instruction bits with *Rs, Rt* and *Imm*.

ADDITIONAL CUSTOMIZABLE FEATURES

In order to guarantee a modular interface for all instruction execution, the architecture implements

[1] Notice that, in order for the instruction's functional specification to be correctly implemented, the user must also specify a custom functional unit capable of executing FOO.

special instructions, such as branching and memory, by using special reservation stations and functional unit pairs, modified to implement the requirements of such instructions. The development of these functional units led to the introduction of additional customization features; while not as easily accessible and modular as the standard functional unit and reservation station parameters, these additional customizations can still be used to implement more complex and efficient functional units.

First, reservation stations can be customized to handle separate commit signals for the arbiter and the CDB. With this method, it is possible to create a functional unit that handles instructions writing to the register file as well as instructions that do not modify register values.

Second, functional units can be optimized to require a variable number of cycles for completing an operation. The standard functional unit setup requires the user to specify a fixed number of cycles after which the operation result will be valid and stay valid, while hidden logic controls the reservation station – functional unit communication. It is, however, possible for the user to introduce an internal "ready" value to signal when the operation has been completed. The API for the *internal_ready* signals is for it to be a single-cycle pulse, having a value of one in the cycle whenever the result is ready. The result need only be valid in the cycle when *internal_ready* is high, while internal logic saves the result and handles communication signals, effectively resulting in variable pipeline depth. As shown in the next section, this variable-latency protocol may be used to optimize performance of complex internal logic, such as dividers.

5- TESTING AND RESULTS

This section describes various tests performed on the processor architecture in order to assess the feasibility for use by hardware developers. To evaluate the practical functionalities of this architecture, I proceed to consider the various processor constraints typically required by an integrated system solution, as well as by the processor user / hardware designer. First, I discuss the issues and tradeoffs in assessing performance and area measurements for the architecture, after which I present the results of restrictive yet unambiguous processor tests for performance and area. Second, I compare the architecture to three commercially available soft processors, all synthesized on the same FPGA, in terms of features, area and performance, and discuss the validity and usefulness of these comparisons. Third, I analyze the costs and benefits of inserting redundant functional units to improve performance. Finally, I analyze a realistic use situation, wherein a hardware designer needs to use the processor for Mp3 decoding though the MADplay library. In particular, the compiled code requires the introduction of a custom fixed-point multiply/divide functional unit, with associated instruction-set extensions.

PERFORMANCE AND AREA

The fundamental, functional constraint for any processor to be useful is adequate performance for the assigned system tasks. Given a certain application and instruction-set architecture (in this case, MIPS-I), such performance requirements can be roughly defined and measured in terms of instructions per second. When constructing an integrated system solution (as opposed to a general-purpose machine), the hardware designer can make reasonable expectations as to which kinds of processor-intensive applications the system will have to run, as well as on a minimum performance requirement: for example, a wireless webcam solution may require JPEG compression and transmission of 640*480 pictures at 10 pictures/second, while a wireless monitor displaying a Blu-ray movie would need to perform H.264 decoding of 1080p images at 32 to 60 frames per second. When such "bottleneck" applications are compiled for a certain instruction set architecture, the designer can obtain a rough estimate of the instructions-per-second (IPS) constraint on the processor.

In order to obtain a general estimate of processor performance, I assembled a simulation tool-chain for running various benchmarking programs on a reference processor implementation. The tool chain performs various tasks to obtain an estimate of processor performance on a given implementation platform. In particular, source code is cross-compiled for MIPS-1 ISA with no coprocessors, then it is linked and assembled. The resulting file is used to generate a Verilog

description of a block-ram, whose initialization contents and locations match the appropriate instruction and data values for the assembled program. The resulting memory is used as a comprehensive instruction/data memory (Von Neumann architecture) and connected to the core. Finally, a processor run with the new memory is simulated. An analysis of simulation logs reveals the number of required cycles and the number of performed operations, thus giving an estimate of Instructions per Cycle (IPC).

In order to obtain the measurement of instructions per second (IPS), the processor needs to be synthesized to estimate the maximum clock frequency which, multiplied by the IPC measurement, results in $\frac{Instructions}{Cycle} \times \frac{Cycles}{Second} = \frac{Instructions}{Second}$. Clock frequency estimation, however, introduces another benchmarking problem. When dealing with a soft-architecture, in fact, the maximum clock frequency depends not only on the internal logic delays of the various processor pipeline stages, but rather on the platform where the processor is implemented. As the target integrated systems are represented by modern FPGAs, the architecture was synthesized for implementation on a Xilinx Virtex-5 chip.

Similarly to clock frequency, the area of a soft processor is determined by the implementation platform. In particular, when mapping soft designs to pre-existing logic, FPGAs typically require a much larger implementation area than equivalent custom integrated circuits. Once again though, since FPGAs are the primary target of the architecture, area constraints were measured for the Xilinx Virtex-5 chip. In particular, three measurements were obtained, two specific to the implementation (percentage of resources utilized, number of FPGA "slices" used), another more general (total number of gates and registers).

Finally, an appropriate benchmark code must be chosen. Initially, the processor was tested on long series of basic "test blocks" for testing the correctness in the execution of each instruction. Subsequently, the processor was tested on Mp3-decoding code taken from the MADplay fixed-point encoding/decoding libraries, a choice explained in more detail later in this section.

Overall, assembling a test platform for a soft architecture presents several challenges and tradeoffs. In general, the choice of all test parameters, from benchmarking code to performance indicator to implementation platform, can dramatically influence any measurement of processor performance and area. However, such choices usually imply a tradeoff between restrictive and ambiguous results. In this case, considering the intended use for application-specific integrated systems, the test performed is very restrictive for a typical implementation and application, and very conservative in terms of implementation assumptions, yet the results are unambiguous. Table 1 presents the test specifications and results for two different processor implementations, a baseline mips-1 compatible version as well as a version featuring an additional fixed-point

divide/multiply functional unit.

Table 1: Processor test results for different benchmark programs and functional unit configurations. Out-of-order execution determines an average IPC (instructions per cycle) of 0.6 on test code, 0.5 on mp3 decoding, and 0.4 on floating-point division code. This variability in performance depends on the increased utilization of the deeply pipelined (32 cycles) MUL/DIV unit; the introduction of the complex MUL/DIV unit also causes a 26% increase in area requirements[2]. The highly-pipelined design allows for a rather high clock frequency of 125-138 MHz, where the unoptimized MUL/DIV unit acts as a delay bottleneck. While the 51-83 MIPS performance measurement is more meaningful, the figure of 614,000 software-implemented floating-point divisions per second is pehaps less ambiguous as a platform-independent measurement.

Compiler	mips-elf-gcc (GNU compiler collection's MIPS-specific compiler), with flags –mips1 (specifying the ISA) and –msoft-float (do not use floating-point coprocessor)	
Test Code	Test 1:	series of instruction-test routines
	Test 2:	series of fixed-point arithmetic subroutines[3] used by MADplay library to reproduce floating-point calculations for Mp3 decoding
Target platform	Xilinx Virtex-5 LX50 FPGA (28,800 slices flip-flops, 28,800 slices look-up tables, 1,728 kb block RAM), approx. equivalent to 3 million gates + RAM.	
Synthesis/Simulation	Xilinx ISE / ISE simulator	
Processor versions	Version 1 [baseline]: 6 RS/FU pairs, implementing the full MIPS-1 ISA Version 2: 7 RS/FU pairs, implementing the full MIPS-1 ISA plus fixed-point multiplication/division, with additional instructions MULT, MULTU, DIV, DIVU, MFHI, MFLO	
Post-synthesis Area (FPGA resources)	Version 1	1801 flip-flop (6% of total), 2848 LUT (9% of total)
	Version 2	2172 flip-flop (7% of total), 3407 LUT (11% of total)
Post-synthesis Area (gates estimate)	Version 1	225,000 gates
	Version 2	285,000 gates (~26% overhead for MUL/DIV unit)

[2] It is important to notice that the MUL/DIV unit is a straightforward, unoptimized implementation of the Hennessy-Patterson algorithm (Hennessy & Patterson, 2006): while the introduction of this unit was focused towards testing flexibility, a performance-oriented implementation would yield much better results in terms of area and latency.

[3] All subroutines included in the fixed-point functions (fixed.c) set of libmad (MADplay's encoding-decoding library), including floating-point addition, subtraction, absolute value and division.

Clock Frequency	Version 1	138.6 MHz (period 7.21 ns)
	Version 2	125.1 MHz (period 7.98 ns)
Instructions per cycle	Test 1	Version 1: 0.61 IPC Version 2: 0.61 IPC
	Test 2	Version 1: not suitable (test required divider unit) Version 2: 0.52 IPC (0.41 IPC for FP divisions only)
Instructions per second	Test 1	Version 1: 83.1 MIPS Version 2: 75 MIPS
	Test 2	Version 1: N/A (requires MUL/DIV unit) Version 2: 63.8 MIPS (51.3 MIPS for FP divisions only)
Floating-point divisions / sec (with one MUL/DIV unit)	Version 2	684k FPDIV/sec (51.3 MIPS @ 75 ops / FPDIV)

COMPARISON WITH EXISTING PRODUCTS

Benchmarking processors is fundamentally a way of comparing their characteristics, thus it would be useful to compare the tested processor versions to existing processors. As the architectural differences between processors widen, however, it becomes increasingly difficult to objectively compare them. While hard processors can be compared in terms of cost, power and performance on a given computer program, soft processors introduce many additional variables, as both area and performance are heavily influenced by the implementation platform and whether the design has been optimized for it. On the other hand, while performance can be ambiguous, commercially available soft processors can be meaningfully compared in terms of architectural features and logic complexity.

Table 2 presents a feature comparison between the customizable processor and three mainstream, commercial synthesizable processors, one of which (Microblaze) is designed and optimized specifically for implementation on Xilinx FPGAs.

Table 2: Feature comparison of the version 2 processor (with added divide / multiply unit) and three commercial synthesizable processors[4]. Even considering the generally high flexibility of all soft processors, a side-by-side comparison reveals the advantages of a customizable functional unit solution. In particular, advantages can be seen in the variable pipeline depth and MUL/DIV latency, in

[4] All data for LEON 2, MicroBlaze and OpenRISC 1200 processors in Table 2, Table 3 reproduced from (Mattsson & Christensson, 2004), authorization pending.

the switchable branch delay slot and in the possibility to adapt custom floating-point units to the architecture. All the additional customization features are allowed by the custom functional unit approach, as opposed to current coprocessor-based solutions. Notice that, out of the processors analyzed, only the OpenRISC 1200 is substantially customizable, supporting custom instructions and coprocessors.

	LEON 2	MicroBlaze	OpenRISC 1200	MIPS Version2 (with divide / multiply unit)
Distributed File Format	VHDL	EDIF	Verilog	Verilog
General				
Architecture	32-bit RISC	32-bit RISC	32-bit RISC	32-bit RISC
Byte ordering	Big Endian	Big Endian	Big Endian	Big Endian
Pipeline depth	5	3	5	Variable (4+)
Issue type	Single	Single	Single	Single
Branch prediction	No	BHB	No	No
Register File				
Organization	Windowed	Flat	Flat	Flat
# of global registers	8	32	32	32
# of windows	2-32	N/A	N/A	N/A
Total # of GPR	40-520	32	32	32
ISA				
Type	SPARC V8	Microblaze	ORBIS32	MIPS
MUL latency (cycles)	1-35	3	3	customizable
DIV latency	35	34	64	customizable
Branch delay slots	1	1	1	variable: 0, 1, 2
Branch latency	0-1	1-3	Unknown	variable: 1 - # of pipeline stages
Load delay	1-2	2	Unknown	1
Custom instruction	No	No	Yes	Yes
Custom coprocessor	No	No	Yes	No
Custom functional units	No	No	No	Yes

	LEON 2	MicroBlaze	OpenRISC 1200	MIPS Version2 (with divide / multiply unit)
Hardware floating-point support	GRFPU, Meiko FPU, LTH FPU	Quixilica FPU	No	Customizable (within FU)[5]
Memory Structure	Harvard	Harvard	Harvard	Harvard / vonNeumann
Memory Management Unit	Yes	No	Yes	No

While the objective of this architecture is better expressed in terms of processor feature comparisons, a quantitative side-by-side comparison in terms of area and performance can give an idea of whether this solution could be feasibly used in the future. Obviously, comparing a completely unoptimized prototype with successful commercial products must yield biased results, yet such a comparison can be useful at least to estimate the degree of improvement and optimization required to obtain a competitive architecture. In Table 3, the architecture was synthesized and simulated for implementation on a Xilinx Virtex-II FPGA (XC2V3000fg676-4), and its area requirements compared to those obtained by (Mattsson & Christensson, 2004) when implementing commercial soft processors on the same platform. Additionally, Mattsson and Christensson present performance results for the three processors running the Dhrystone 2.1 benchmark (Weicker, 1984). Originally developed as a compact comprehensive benchmarking software for integer performance, the Dhrystone program consists of a main loop with a clock and a counter tracking the number of iterations, producing a measure of program iterations per second to rank processor performance. This integer based, platform-independent approach makes the Dhrystone a suitable benchmark for embedded systems. On the other hand, with the Dhrystone being a short, open-source and widely spread program, its results can be heavily distorted through compiler optimization, resulting in many processor architectures advertising unrealistically high results. For these reasons, Dhrystone is currently considered unreliable for modern architectures, yet its compactness and low utilization of operating system calls makes it the most feasible choice for comparisons. Additionally, data is available for other processors that were tested by independent researchers and without excessive compiler optimizations, thus diminishing result distortion.

[5] The possibility to insert a custom FPU as a functional unit with dedicated instructions is discussed in the customization test section.

Table 3: Area, Frequency and Dhrystone benchmark performance comparison of the version 1 and version 2 processors with three commercial synthesizable processors. The three commercial processors are analyzed in two different configurations, optimized respectively for high performance and for low area, while both MIPS versions are measured in a single, unoptimized implementation. Area measurements are given as total number of 4-input lookup table slices used on the FPGA for the comprehensive system (core plus cache[6]). Both versions of the MIPS present lower area and higher clock frequency than both the LEON 2 and OpenRISC 1200 regardless of configuration, while the MicroBlaze displays more compactness and lower latency than all other architectures. This specific comparison, however, is heavily biased towards the MicroBlaze, which has been specifically developed and optimized by Xilinx to run natively on Virtex-series FPGAs, while all other solutions are designed in a portable, more general Verilog/VHDL format. On the other hand, post-synthesis measurements for MIPS clock frequency, while generally conservative, are less reliable than post-PAR figures given for the three commercial processors. On the Dhrystone benchmark side (estimate[7]), the MIPS is outperformed by most configurations. The benchmark deficiencies are due to two factors: low IPC efficiency, which can be improved by inserting redundant functional units, and low compiler efficiency due to a restrictive instruction set, which can also be expanded by inserting additional functional units and instructions, conforming to more advanced MIPS ISAs (MIPS-2, MIPS32 etc.), as well as by compiler optimization.

	LEON 2	MicroBlaze	OpenRISC 1200	MIPS Version2 (with divide / multiply unit)	MIPS Version1
Platform	Xilinx Virtex-II XC2V3000fg676-4				
HW resources	28,000 slices flip-flops, 28,800 slices 4-input LUT				
Area (LUT)					
Performance-optimized	8794	2442	6443	4579	3037
Area-optimized	5871	2325	5865		
Clock Frequency (MHz)					
Performance-optimized	53.3	80	40	78	91
Area-optimized	26.7	26.7	26.7		

[6] Including the cache in the area measurements does not compromise the results, since RAM and cache modules on the Virtex-II are entirely implemented using Block-RAM slices, which do not add to the LUT count. While this measurement may bias performance/area measurements in favor of systems with a larger cache, the compactness of Dhrystone code does not give an advantage to cache-heavy implementations.

	LEON 2	MicroBlaze	OpenRISC 1200	MIPS Version2 (with divide / multiply unit)	MIPS Version1
Dhrystone 2.1 iterations/sec					
Performance-optimized	78431.4	76189.6	26454.9	6380 – 23790 (estimated[7])	N/A
Area-optimized	30690.5	23188.3	10653.8		
Performance/ Area ratio					
Performance-optimized	8.91	31.20	4.10	1.39 – 5.19 (estimated)	N/A
Area-optimized	5.22	9.97	1.81		

INCREMENTAL PERFORMANCE AND COST FOR REDUNDANT FUNCTIONAL UNITS

As previously discussed, one limitation of the processor architecture is that functional units require at least two cycles to perform an operation, since reservation stations cannot be loaded while in write back state. While for a 32-cycle divider unit this limitation does not influence performance much, combinational units for common instructions (add, sub etc.) are deeply penalized by the extra cycle latency, effectively limiting processor performance to 0.5 IPC for all series of instructions mapped to the same unit. This bottleneck suggests that adding redundant functional units can improve performance, yet the feasibility of this method relies on the costs and benefits of inserting an additional functional unit. It is intuitive that an additional combinational FU can double performance on a long series of instructions mapped to a common FU, yet showing

[7] Running Dhrystone benchmark is still in-progress. The procedure for the estimate is as follows. Considering that all other processors are single-issue, and conservatively assuming that they run at an ideal one instruction per cycle, their IPS becomes equal to their clock frequency. Dividing clock frequency by Dhrystone score (iterations/second) produces $\frac{Instructions}{Second} \div \frac{Iterations}{Second} = \frac{Instructions}{Iteration}$. This calculation yields a conservative (max values) range of 679-2506 instructions per Dhrystone iteration. Considering its restrictive instruction set and consequently lower compiler efficiency, a range estimate for the MIPS is 2000-5000 instructions per iteration. Multiplying by the known clock frequency and a 0.4-0.6 range of IPC yields $[78M]\frac{Cycles}{Second} \times [0.4\ to\ 0.6]\frac{Instructions}{Cycle} \div [2000\ to\ 5000]\frac{Iterations}{Second} = [6240\ to\ 23400]\ \frac{Instructions}{Iteration}$

this would not be a meaningful example. As a more realistic test, Table 4 and Figure 7 present cost/benefit results for the insertion of an additional ADD/SUB unit for a vector add/subtract code, showing that inserting redundant functional units can be highly advantageous.

While Table 4 shows very low area and latency costs for additional functional units, such costs are very likely to increase with the total number of units. As previously discussed, this is due to the wiring complexity of the Common Data Bus system for result broadcasting, wherein essentially all reservation stations must be connected. It is thus evident that, while a seventh FU must connect to only six pre-existing elements, each additional FU will imply a incrementally higher complexity, which will influence both incremental area (number of wires) and performance (CDB latency). Finally, as the number of functional units exceeds powers of two, all FU indexing wires will expand by one bit, thus implying a one-time area and latency cost, which however should be minimal. While incremental cost issues could be explored further, it is reasonable to assume that, in its typical configuration, the processor will not feature a high number of additional functional units, thus avoiding such uncertainties.

Table 4: Area cost and performance benefit for insertion of a redundant ADD/SUB unit in vector addition/subtraction code. Notice that the insertion of a simple functional unit implies a minimum incremental costs for both area (+4%) and frequency (-1.5%). The +25% area figure for the insertion of the DIV/MUL unit in Table 1 is thus due to the high complexity and latency of the unit, and is not representative of the costs of expansion. Considering the low complexity of an ADD/SUB unit, the cost figures presented are a good measurement to the area cost of reservation station / FU machinery, and the incremental delay cost of the higher wiring complexity of the Common Data Bus for additional units.

	MIPS Version1 (6 FUs)	MIPS with redundant ADD/SUB unit (7 FUs)
Platform	Xilinx Virtex-5 LX50 FPGA	
Area	1801 FF, 2743 LUT	1876 FF(+4.1%), 2875 LUT (+4.8%)
Clock Frequency	125 MHz	123 MHz (-1.5%)
Instr. per cycle	0.7	0.81
Performance (MIPS)	87.5	99.6 (+13.8%)

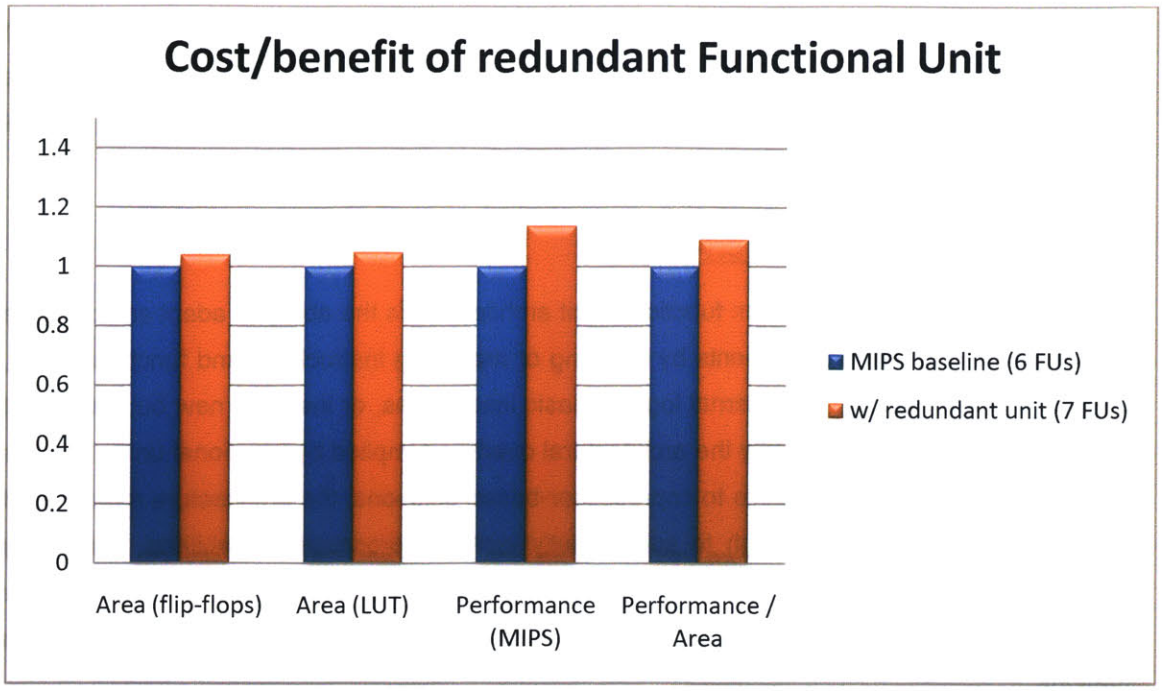

Figure 7: Relative cost and benefit of inserting a redundant ADD/SUB unit for vector addition / subtraction code. It is evident from the graph that performance benefits of inserting the redundant unit outweigh area costs for this particular test.

LIMITATIONS AND COMPILER CONSTRAINTS

While a major advantage of this architecture is the possibility to define custom instructions, an equivalent necessity is for compiled programs to utilize the additional features. In particular, compiler support is required to generate programs that take full advantage of the processor's features -- a drawback shared with coprocessor-based architectures. The development of a flexible compiler (dynamically and optimally mapping code to a variable set of instructions) is a fascinating academic field, yet it is an excessive request to make for supporting processor flexibility. However, in the context of dedicated embedded systems, it is expected that program compilation be optimized for the specific implementation platform, and also that highly-used assembly-level subroutines (such as MADplay's floating-point algebra library) be specifically tailored for high performance on the target system. In this context, users can modify assembly-level subroutines to take advantage of any additional custom instruction, thus producing efficient (though sub-optimal) code that takes advantage of architectural flexibility without requiring flexible compilers. Similarly, many common assembly-level optimization techniques can be adapted to generate programs for custom architectures. While functional units and instructions within the processor can be modified very rapidly, rewriting entire subroutine libraries is a complex task, thus eliminating the possibility of recursively modifying the processor's instruction set for

obtaining optimal performance. On the other hand, internal functional unit design, instruction mapping and insertion of redundant functional units can all be modified independently of source code, thus still opening the option to recursively reconfigure the processor for optimal performance.

ADAPTABILITY IN A REALISTIC SITUATION

The fundamental virtue of a custom functional unit architecture is the ability to adapt and optimize a processor for different environments by inserting or modifying instructions and functional units. However, simply modifying the internal logic of basic instructions, or inserting new but redundant instructions is not enough to justify the architectural overhead implied by functional units. In order to be considered a valid alternative to coprocessor-based solutions, the architecture must exhibit a degree of flexibility high enough to substantially increase processor capabilities, just like a coprocessor can, but without the need for a dedicated coprocessor interface. Furthermore, the flexibility must not be exhibited only within ad-hoc testing environments, but rather in real design situations.

Libmad and Mp3 decoding

In order to evaluate architectural flexibility in a realistic situation, the processor was tested in a typical task for current embedded systems: Mp3 decoding. Out of several available options, the MADplay encoding / decoding library (Underbit Technologies, 2005) was chosen, for several reasons. First, the MADplay library (libmad) uses only fixed-point computation, thus making it suitable for dedicated processors with simplified instruction sets. Also, the library is distributed as free software, highly modifiable and optimized for gcc-based compilers, thus making it an excellent choice for portable solutions. Finally, the library is a common choices for many popular software products, both portable and desktop-based (GSPlayer, MPlayer, OpenRISC), thus constituting a very likely choice for an embedded solution.

Using the entire library and a complete mp3 as a test case, however, resulted infeasible, since the code being too long to be simulated as an instruction memory, and the front-end functions is excessively relying on hard to compile OS calls, which would be modified in an embedded system implementation. The most frequent and computationally intensive function calls within libmad are a series of integer arithmetic subroutines that are used to reproduce floating point operations: compactness, complexity and exemplarity made this series of floating point functions an adequate choice for a sample code to be used for testing.

Divide and multiply: special instructions introduced as new functional units

The series of arithmetic subroutines used as a test case requires utilization of the full MIPS-I ISA, including fixed-point multiply and divide instructions, which were not originally implemented within the architecture. Differently from all other arithmetic functions, the MIPS implementation of divide and multiply requires architectural modifications to the processor, including the introduction of two special-purpose 32-bit registers, *Hi* and *Lo*, used respectively to store quotient and remainder for a division, or a 64-bit product for multiplication. Additionally, the divide and multiply instructions need not write to the register file but only modify the special-purpose registers. Finally, for communicating results, the instructions MFHI and MFLO (move from hi/lo) are used to write the content of a special register to the register file.

Since the implementation of special-purpose registers and instructions would normally require architectural modifications, as a realistic flexibility test a divide/multiply unit with related instructions was implemented by solely accessing the customizable functional unit / instructions interface. In particular the additional customization features previously described, allowing variable pipelining depth and switchable access to the register file, were developed to increase functional unit flexibility in order to allow the introduction of special function units. Figure 8 illustrates in detail the implementation of the multiply/divide unit, capable of processing the special instructions by instantiating special-purpose registers internally, and by using an FPGA built-in multiplier as well as an externally-defined Hennessy-Patterson divider.

The highly customizable architecture allowed for a special-instruction functional extension of the processor, which made it correctly comply with an extended ISA to decode Mp3 files. This example of extending processor functionality by inserting new instructions and functional units indicates how the architecture can be externally adapted by the user to comply with different tasks and requirements. Extending the example, the insertion of a hardware floating-point unit or multimedia unit, both traditionally implemented through a coprocessor, could be performed by introducing new functional units with related instruction. Additionally, one would be able to design the internal functioning of such modules, or adapt an external pre-existing unit (as was the divider), without having to be constrained by compatible processor-coprocessor APIs.

Example Special Functional Unit: MUL/DIV

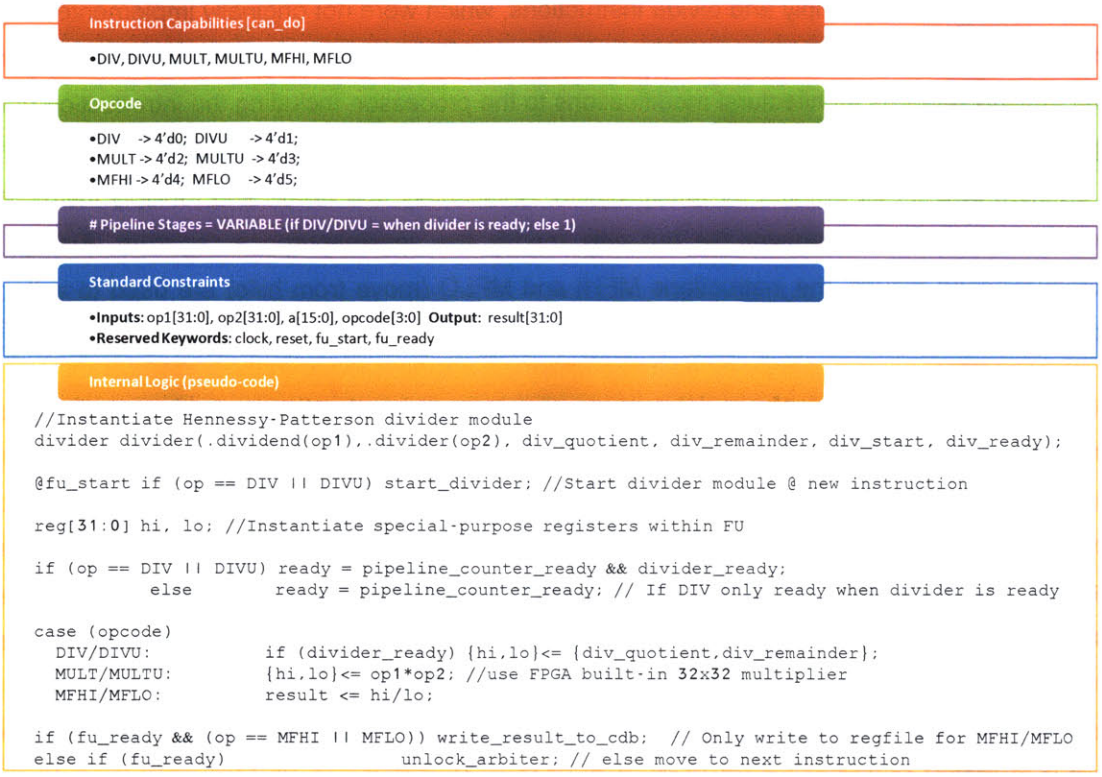

Figure 8: Custom functional unit implementing special instructions to perform fixed-point multiply and divide. By looking at the pseudo-code, it is evident that custom functional units feature the same internal logic flexibility that a coprocessor may enjoy. In this case, the special-purpose registers *hi* and *lo* required to implement the instructions are instantiated and accessed exclusively within the functional unit. Multiply and divide instructions only change the internal special registers, to the register file; a separate divider module is instantiated for handling integer divisions; for divisions, the pipelining latency of the functional unit is flexible, interfacing with the divider to determine when the operation has been completed; the result field always has a value, but the value is written to the register file only for MFHI/MFLO instructions. Such powerful flexibility options may be used to develop more complex functional units and related instructions, effectively allowing functional extensions of coprocessor-level complexity without modifying the architecture.

6- SUMMARY: CONTRIBUTIONS AND FUTURE WORK

In this thesis, I set out to solve a problem: develop a highly flexible microprocessor architecture by borrowing selected technologies from the high-performance computing domain. The objective of my work is to achieve a superior architectural alternative to coprocessor-based solutions, by providing the user with the possibility to create or reconfigure a variety of processor features in a fully modular manner. In this final section, I set out to outline my contributions towards solving the problem, as well as a list of future works that I believe can improve on my results.

My main contributions towards a solution are:

- ✓ *Evaluating and selecting a combination of high-performance technologies,* borrowing and adapting these techniques to achieve a flexible yet balanced microprocessor architecture. In particular, I opted for out-of-order processing, modular and parallel functional units, as well as in-order instruction commit, while also developing new techniques for achieving a coherent system.
- ✓ *Designing and implementing a soft processor architecture* that combines the selected technologies while conforming to the popular MIPS instruction set. I developed, simulated and tested the prototype as a Verilog design compatible with the MIPS-1 ISA.
- ✓ *Developing a modular customization interface* by which the user can insert or modify instructions and functional units. The interface provides a large variety of customization parameters, ranging from variable internal pipeline depth to internal state within functional units, free choice of instruction operands, destination, register file behavior and other control signals, variable number of functional units and insertions of redundant units for performance improvement.
- ✓ *Testing the prototype on a variety of realistic situations for flexibility and feasibility of use, also comparing it to commercially available embedded processors.* First, I obtained estimates of processor area, frequency and performance on various implementation platforms and for various configurations. Second, I presented a side-by-side comparison of the architecture with commercial alternatives, comparison that highlights many advantages in terms of flexibility and customization, comparable or superior results for area and latency, and some deficiencies within performance benchmarks. Third, I analyzed the costs and benefits of adding redundant functional units for improved performance. Finally, I stressed the limits of architectural flexibility by testing the prototype on algebraic subroutines from an Mp3-decoding library, extending the architecture by introducing fixed-point divide / multiply capabilities through the

customization interface. This test suggested that the customization interface provides sufficient flexibility for introducing coprocessor-equivalent functional units.

These combined contributions demonstrate that a modular, functional unit based architecture can be successfully introduced as a valid alternative to coprocessor-based APIs. When considering the initial objective of providing a superior alternative to existing architectures, however, it becomes evident that a few more issues need to be addressed for the solution to be deemed complete. Each of these issues, however, has potential to be addressed by future works, for some of which the bases have been laid during the development of this project.

- *Achieving higher baseline performance while retaining flexibility* is a very feasible objective. In particular, performance fallacies such as low IPC are due to a lack of optimization rather than to architectural deficiencies, and since the project's focus was not to achieve high performance, lots of room is available for improvement. First, a mechanism can be developed that allows combinational functional units to achieve full utilization, which in terms of performance would be equivalent to a complete set of redundant functional units (with high performance benefits as shown by the tests). In the development of the processor, I laid the bases of such mechanism, which however has remained unimplemented. Second, the low area and delay of the prototype leave enough resources for implementing a larger baseline ISA (i.e. MIPS-2 or higher), which in turn would increase compiler efficiency and performance. Third, experimenting various baseline processor configurations, in terms of area, instruction distribution, number of functional units and comparative delays would yield a more efficient implementation. Particularly, with the introduction of slow, complex units such as DIV/MUL, it may be convenient for the single-cycle instructions to be implemented in fewer, more complex functional units.
- *Introducing an XML-based customization interface* would further simplify user interactions. Currently, customization is implemented within Verilog code, through a highly modular editable source code library. It would be useful to further simplify the user experience by specifying XML-based processor descriptions in turn used to generate Verilog. The bases for this improvement have been laid by the REDSOCS group, by means of a series of Python scripts that read XML-based processor descriptions, generating the resulting Verilog design by modifying the baseline version (currently under development).
- *Performing more extensive benchmarking* would be useful to evaluate the real-world feasibility of the architecture. The complexity of cross-compiling for custom embedded processors and the difficulties of simulating execution limited the

benchmarking results for this thesis, yet benchmarking the processor on a complete, modern application would be highly useful. Again, a tool chain for compiling source code, developed and currently used by the REDSOCS group, provides a good starting point for the setup of a complete benchmarking environment.

- *Developing a dynamic compiling tool and a recursive optimizer* would dramatically improve the utility of the architecture. While both very ambitious and not yet explored, these two solutions combined could in theory allow the exploration of unsupervised processor configuration for target applications, a different yet interesting perspective on performance optimization.

7-APPENDIX

PRELIMINARY EXPERIMENT

When initially considering the task of developing a flexible MIPS architecture, some preliminary experiments were performed, with the aim of assess whether a highly flexible architecture can be designed without carrying excessive hardware complexity. In the spring of 2007, in collaboration with Olivier Bichler and Alessandro Yamhure, I designed an experimental MIPS processor with variable-depth pipelining (Bichler, Carli, & Yamhure, 2007). Implemented in Bluespec, the processor works on the concept of guarded atomic actions, wherein different pipeline stages are implemented as independent logic blocks (referred to as *rules*), communicating through FIFO buffers rather than registers. A set of logic checks (*guards*) ensures that rules being executed in parallel do not have reciprocal data dependencies, thus guaranteeing absence of data hazards. The design is customizable by the user who, at design-time, can choose between three implementations with varying pipeline depths.

Area and performance results for various processor implementations outlined important learning points. First, it was clear that the Bluespec language is not an ideal developing platform for a highly customizable processor. In particular, while enforcing correctness in parallel processing, the atomic actions structure does not conform well to architectures with highly interconnected components, being mostly useful for very linear processes. As a result, the flexible architecture resulted much larger and more complex than its equivalent MIPS baseline. On the cost/performance side, however, it was evident that the relative changes in performance, latency and area between various configurations can give origin to balanced tradeoffs. This result suggested that each implementation across a highly flexible architecture may be optimal for a particular task, with no configuration being dominated by another across the entire measurement spectrum.

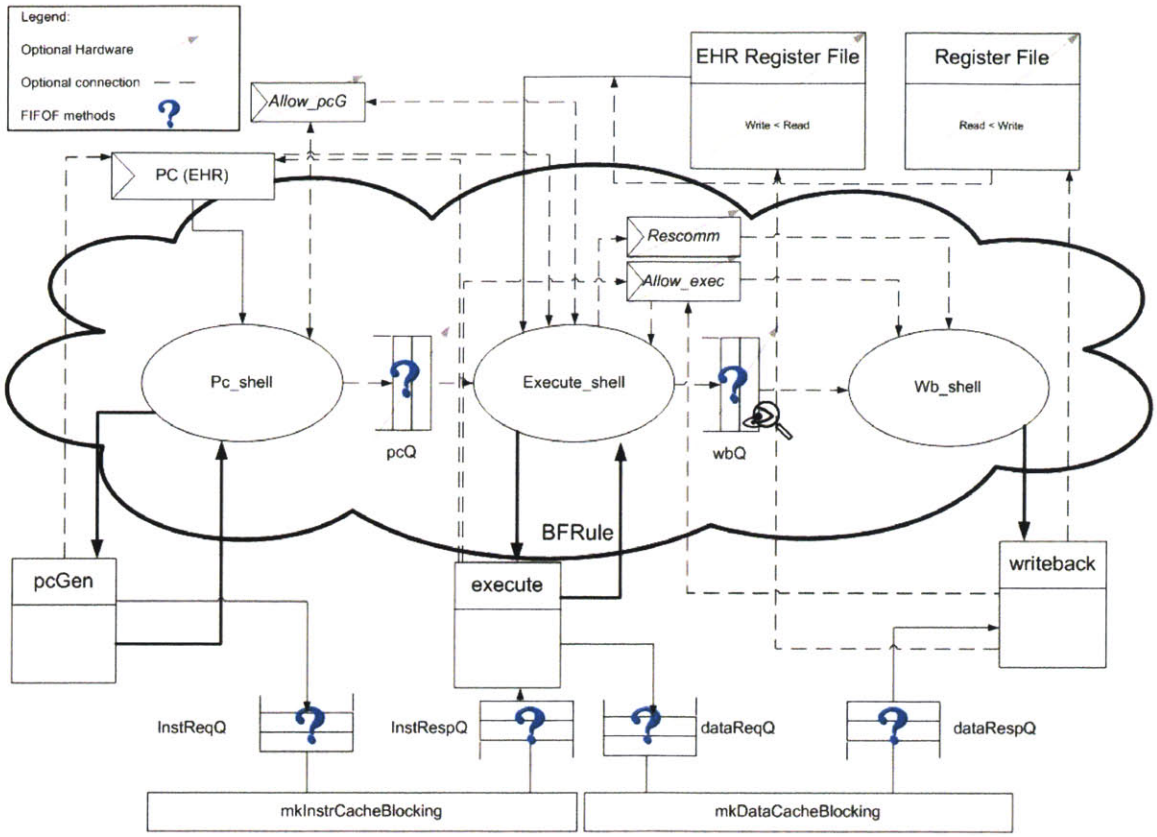

Figure 9: High-level schematics for a Bluespec-based MIPS architecture with variable-depth pipeline. Dotted lines and arrows represent connections that are introduced or deleted according to the processor version. The baseline, 3-stage processor is implemented as three atomic rules, *pc, execute* and *writeback*. The rules communicate to each other and the memory through FIFO buffers, which ensure independent reads and writes. Each rule is further divided into a *shell* (ovals), customized components that take care of I/O assignments and data dependencies, and *functions* (rectangles), purely combinational circuitry handling the actual instruction execution. The specification of a pipeline depth parameter causes the merge or separation of atomic rules, and consequent introduction or elimination of FIFO buffers, thus allowing a choice of combinational, 2-stage or 3-stage processor implementations.

8- REFERENCES

Anderson, E. C., Svendsen, H. B., & Sohn, P. A. (1996). *Patent No. 5577250.* United States of America.

Bichler, O., Carli, R., & Yamhure, A. (2007). *A Parametrizable Processor.* Cambridge MA: Dept. of Electrical Engineering and Computer Science, M.I.T.

Fisher, J. A., & Freudenberger, S. M. (1992). Predicting conditional branch directions from previous runs of a program. *ACM SIGPLAN notices* , 85-95.

(2002). System-Level Modelling for Performance Estimation of Reconfigurable Coprocessors. In M. Glesner, P. Zipf, & M. Renovell, *Field-programmable Logic and Applications: Reconfigurable Computing is Going Mainstream* (pp. 567-576). Montpellier: FPL.

Hauser, J., & Wawrzynek, J. (1997). Garp: a MIPS processor with a reconfigurable coprocessor. *5th IEEE Symposium on FPGA-Based Custom Computing Machines (FFCM '97)*, (p. 12).

Hennessy, J. L., & Patterson, D. A. (2006). *Computer Architecture: A Quantitative Approach* (4th Edition ed.). Morgan Kaufmann.

Kapre, N., Mehta, N., deLorimier, M., Rubin, R., Barnor, H., Wilson, J., et al. (2006). Packet Switched vs. Time Multiplexed FPGA Overlay Networks. *14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FFCM'06)*, (pp. 205-216).

Mattsson, D., & Christensson, M. (2004). Evaluation of synthesizable CPU cores. Gothenburg, Sweden: Chalmers University of Technology.

McFarling, S., & Hennessey, J. (1986). Reducing the cost of branches. *International Symposium on Computer Architecture* (pp. 396-406). Tokyo, Japan: IEEE Computer Society Press.

Palmer, J. (1980). The Intel 8087 numeric data processor. *International Symposium on Computer Architecture* (pp. 174-181). La Baule, United States: ACM.

Rudd, K. W. (1997). *Efficient Exception Handling Techniques for High-Performance Processor Architectures.* Stanford CA: Computer System Laboratory, Stanford University.

Tomasulo, R. M. (1967). An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development* .

Tyson, G. (1994). The effects of predicated execution on branch prediction. *Microarchitecture* , 196-206.

Underbit Technologies. (2005). *MAD: MPEG Audio Decoder.* Retrieved April 29, 2008, from www.underbit.com: http://www.underbit.com/products/mad/

Weicker, R. P. (1984). Dhrystone: a synthetic programming benchmark. *Communications of the ACM , 27* (10), 1013-1030.